Corrigé TP9

On utilise dans ce corrigé les fonctions des modules Vector et Heap, à vous de faire la traduction si vous avez choisi de faire le TP avec taquin_full.ml. On fournit avec ce document le code correspondant.

5. C'est clair une fois qu'on a remarqué que pour une valeur v et une configuration c, $|c_v^i - \lfloor v/n \rfloor|$ mesure l'écart vertical entre la position de la case portant le numéro v dans la configuration c et sa position dans la configuration finale ; l'autre terme mesurant l'écart horizontal.

   Ainsi, la somme $h(c)$ vaut zéro si et seulement si la configuration est finale et un déplacement déplace une case de une unité selon l'un des deux axes donc modifie la valeur de l'heuristique en lui ajoutant ou soustrayant 1. On en déduit que, si c et c' sont deux configurations voisines,

   $$h(c) = 1 + (h(c) - 1) \leq 1 + h(c')$$

   ce qui montre que h est monotone puisque le poids de toutes les arêtes du graphe des configurations vaut 1. Comme on a de plus $h(f) = 0$ avec f la configuration finale, h est admissible.

10. Il faut juste faire attention à générer un état indépendant pour chaque déplacement possible à l'aide de la fonction copy. La fonction apply_moves applique chacun des déplacements de la liste qu'est son argument à l'état considéré ; on l'utilise évidemment avec la liste fournie par possible_moves.

```
let successors state =
  let rec apply_moves moves =
    match moves with
    | [] -> []
    | m :: ms ->
      let s = copy state in
      apply s m;
      s :: apply_moves ms in
  apply_moves (possible_moves state)
```

11. Pas de différence fondamentale avec les fonctions de reconstruction de chemin lorsque le stockage des parents se fait dans un tableau. L'argument path de la fonction read_path sert d'accumulateur permettant de reconstruire le chemin directement dans le bon sens. On peut choisir de calculer un chemin de x vers la racine mais il faut penser alors à l'inverser en fin d'algorithme avec List.rev.

```
let reconstruct parents x =
  let rec read_path v path =
    let p = Hashtbl.find parents v in
    if p = v then v :: path
    else read_path p (v :: path) in
  read_path x []
```

12. On commence par définir l'exception No_path. Puis, on traduit exactement l'algorithme du cours à la différence près que les structures de stockage sont des dictionnaires et non des tableaux.

    Lorsqu'on découvre le sommet but (correspondant à la configuration finale), en plus de renvoyer un chemin de longueur minimale permettant de l'atteindre, on affiche la longueur de ce chemin et le nombre de configurations qu'on a du examiner pour le trouver (ce qui correspond au nombre de sommets pour qui A* a pu déterminer un parent donc à la taille du dictionnaire parents) en prévision de la question 13. Remarquer qu'il est plus efficace de tester si la configuration x est finale via x.h = 0 que via h = final (une comparaison à la place de 15).

La fonction Hashtbl.find_opt renvoie Some v où v est l'élément associé à celui recherché si l'association existe et None sinon mais on aurait aussi pu faire une simple disjonction de cas.

```
exception No_path

let astar initial =
  let dist = Hashtbl.create 10 in
  let parents = Hashtbl.create 10 in
  Hashtbl.add parents initial initial;
  let q = Heap.create () in
  Heap.insert q (initial, initial.h);
  Hashtbl.add dist initial 0;
  let rec loop () =
    match Heap.extract_min q with
    | None -> raise No_path
    | Some (x, _) when x.h = 0 ->
      let c = reconstruct parents x in
      Printf.printf "Length: %d, explored states: %d\n"
        (List.length c - 1)
        (Hashtbl.length parents);
      reconstruct parents x
    | Some (x, _) ->
      let dx = Hashtbl.find dist x in
      let process v =
        let dv = dx + 1 in
        match Hashtbl.find_opt dist v with
        | Some d when d <= dv -> ()
        | _ ->
          Hashtbl.replace dist v dv;
          Heap.insert_or_decrease q (v, dv + v.h);
          Hashtbl.replace parents v x in
      List.iter process (successors x);
      loop ()
  in loop()
```

13. Pour répondre à la question, on utilise les affichages qu'on a pris soin d'ajouter à astar ci-dessus. On obtient les résultats suivants (pour les temps d'exécution, on pourrait les calculer précisément au besoin, on se contente ici d'une estimation à l'oeil. Les temps peuvent légèrement varier selon la machine utilisée mais on devrait avoir des ordres de grandeur similaires).

Nom de la grille de taquin	ten	twenty	thirty	forty	fifty
Temps de calcul	instantané	instantané	0.5s	10s	5min
Nombre de configurations explorées	43	1180	11676	354151	4587604

Pour sixty_four, la recherche avec astar échoue au bout d'une dizaine de minutes faute de mémoire.

Remarque : Pour chacun des exemples fournis par l'énoncé, le nom de l'exemple indique la longueur d'un chemin minimal entre la configuration décrite par l'exemple et la configuration finale. Evidemment, il faut vérifier que c'est bien le cas dans votre implémentation pour s'assurer du bon fonctionnement de astar, au moins sur ten dont la construction montre sans doute possible que la configuration finale s'obtient en 10 étapes !

Pour obtenir la suite de mouvements à effectuer pour twenty, il suffit de la retrouver à partir de la suite de configurations fournie par astar. On obtient : Down, Left, Up, Right, Right, Down, Down, Right, Up, Left, Down, Left, Left, Up, Right, Right, Up, Right, Down, Down. On constate qu'il faut bien 20 mouvements pour résoudre ce taquin et on applique ces mouvements à la grille twenty pour vérifier qu'en déplaçant la case vide selon cette séquence, on atteint effectivement la configuration finale.

14. On montre par récurrence sur $m-p$ que DFS(m, e, p) renvoie vrai si et seulement si on peut passer du sommet e au sommet but via un chemin de longueur inférieure à $m-p$. Si $m-p=0$, l'algorithme renvoie vrai si e est est le sommet but et faux sinon car tous les appels sur les voisins de e échouent alors ; d'où l'initialisation.

    Pour l'hérédité, on constate que DFS(m, e, p) renvoie vrai si et seulement si e est le but ou l'un des DFS$(m, x, p+1)$ avec x successeur de e renvoie vrai. Par hypothèse de récurrence, la deuxième condition équivaut au fait qu'il y a un chemin de longueur au plus $m-p-1$ entre x et le but donc dans tous les cas un chemin de longueur au plus $m-p$ entre e et le but.

    On spécialise le résultat obtenu par récurrence avec $e=s$ et $p=0$ pour conclure.

15. Dans les deux cas, la complexité en espace de DFS est en $O(n)$ puisque seule est stockée la pile d'appels représentant le chemin actuellement testé dont la taille maximale est la distance entre le sommet initial et le sommet final. De plus :

    - Dans le premier cas, DFS$(k, s, 0)$ s'exécute en $O(k)$ et comme on appelle successivement DFS$(0, s, 0)$, ..., DFS$(n, s, 0)$, la complexité temporelle de IDS est en $O\left(\sum_{k=0}^{n} k\right) = O(n^2)$.

    - Dans le second cas, DFS$(k, s, 0)$ s'exécute en temps $\sum_{p=0}^{k} 2^p \leq 2^{k+1}$ puisqu'il va explorer tous les états à profondeur au plus k. En sommant le coût des appels pour $k \in [\![0, n]\!]$, on obtient une complexité pour IDS en $O(2^n)$.

    Dans le premier cas, utiliser IDS n'a aucun intéret car un parcours en largeur se ferait en temps et en espace en $O(n)$. Dans le deuxième cas en revanche, un parcours en largeur aurait une complexité temporelle similaire mais une complexité spatiale exponentielle. Autrement dit, si le facteur limitant est la mémoire, l'algorithme IDS devient alors très intéressant.

    Remarque : On peut avoir l'impression que IDS devrait avoir une complexité temporelle rédhibitoire par rapport à un parcours en largeur dans le second cas puisqu'on recalcule fréquemment les "morceaux de chemins" proches de la source. En fait, ces recalculs ne coûtent pas très cher devant les calculs exigés par ceux effectués à la profondeur n (qui eux ne sont pas beaucoup répétés), de la même façon que dans un arbre binaire complet, il y a à peu près autant de noeuds entre les étages 0 et $n-1$ qu'il n'y en a à l'étage n. Pour plus de détails sur cette intuition, vous pouvez vous référer à Intelligence artificielle, Stuart Russel et Peter Norvig, page 94 dans la 3ème édition française.

16. L'exception Found permet de déterminer si le but a été atteint lors d'une itération et de stocker la profondeur à laquelle cette découverte a été faite.

```
exception Found of int
```

    La fonction opposite indique quelle est la direction opposée à celle donnée en argument : elle permet d'annuler un coup (ce qui est utile dans idastar puisqu'on y gère une configuration qu'on fait évoluer au fur et à mesure et qu'il faut pouvoir revenir sur ses pas).

```
let opposite = function
  | L -> R
  | R -> L
  | U -> D
  | D -> U
  | No_move -> No_move
```

Il ne reste plus qu'à traduire le pseudo-code fourni en Ocaml :

```
let idastar_length initial =
  let state = copy initial in
  let rec search depth bound =
    if depth + state.h > bound then depth + state.h
    else if state.h = 0 then raise (Found depth)
    else
      let minimum = ref max_int in
      let make_move direction =
          apply state direction;
          minimum := min !minimum (search (depth + 1) bound);
          apply state (opposite direction); in
      List.iter make_move (possible_moves state);
      !minimum in
  let rec loop bound =
    let m = search 0 bound in
    if m = max_int then None
    else loop m in
  try
    loop state.h
  with
  | Found depth -> Some depth
```

17. Montrons que dans la fonction précédente, si search 0 m renvoie b sans lever d'exception alors la distance d entre le sommet initial et le sommet but vérifie $d \geq b$. Supposons par l'absurde qu'il existe un chemin $C = s_0, ..., s_k$ entre le sommet initial et le sommet but tel que $k < b$.

    Comme search n'a pas levé d'exception, ce chemin n'a pas été intégralement exploré et il existe donc p tel que s_p est le dernier état exploré par search 0 m sur le chemin. L'appel récursif search p m qui a été effectué lorsque state valait s_p s'est donc arrêté en renvoyant $p + h(s_p)$ puisqu'on n'a pas exploré s_{p+1}. On en déduit que $b \leq +h(s_p)$ puisque l'appel racine renvoie le minimum des valeurs de retour des appels récursifs.

    Or la distance de s_p au but est inférieure à $k - p$ par définition du chemin C donc l'admissibilité de h montre que $p + h(s_p) \leq p + k - p = k$ et donc $b \leq k$ ce qui est une contradiction avec $k < b$.

    On en déduit l'invariant suivant pour loop : la distance entre initial et le but est supérieure ou égale à bound (qui est l'argument de loop). C'est initialement vrai puisqu'on part de bound $= h$(initial) et que l'heuristique h est admissible. Cela reste vrai ensuite d'après ce qu'on vient de montrer. L'exception n'est levée que si on trouve un chemin vers l'état final de longueur inférieure à bound qui est donc forcément optimale d'après ce qui précède.

18. On utilise en fait le vecteur path comme on utiliserait une pile. On effectue effectue la recherche en tenant compte du dernier déplacement effectué pour ne pas l'annuler au coup suivant : initialement

le déplacement effectué est No_move, dont on voit à présent l'utilité. On calcule au passage le nombre de sommets visités lors de l'algorithme grâce à counter.

```
let idastar initial =
  let counter = ref 0 in
  let state = copy initial in
  let exception Found in
  let path = Vector.create () in
  let rec search depth bound last_move =
    incr counter;
    if depth + state.h > bound then depth + state.h
    else if state.h = 0 then raise Found
    else
      let minimum = ref max_int in
      let make_move direction =
        if direction <> opposite last_move then (
          Vector.push path direction;
          apply state direction;
          minimum := min !minimum (search (depth + 1) bound direction);
          apply state (opposite direction);
          ignore (Vector.pop path)
        ) in
      List.iter make_move (possible_moves state);
      !minimum in
  let rec loop bound =
    let m = search 0 bound No_move in
    if m = max_int then None
    else loop m in
  try
    loop state.h
  with
  | Found ->
    Printf.printf "%d node expansions\n" !counter;
    Some path
```

19. On parcourt le vecteur renvoyé par idastar en appliquant string_of_direction à chaque élément pour transformer la suite de déplacements sur un plus court chemin en suite de chaînes de caractères. La fonction print_idastar affiche en plus de la suite de déplacements minimale, sa taille (pour vérification). Il ne reste plus qu'à exécuter print_idastar fifty.

```
let string_of_direction = function
  | U -> "Up"
  | D -> "Down"
  | L -> "Left"
  | R -> "Right"
  | No_move -> "No move"
```

```
let print_direction_vector t =
  for i = 0 to Vector.length t - 1 do
    Printf.printf "%s " (string_of_direction (Vector.get t i))
  done;
  print_newline ()

let print_idastar state =
  match idastar state with
  | None -> print_endline "No path"
  | Some t ->
    Printf.printf "Length %d\n" (Vector.length t);
    print_direction_vector t
```

20. La fonction astar échoue à trouver une solution à sixty_four faute de mémoire. La fonction idastar en revanche trouve une solution (qui est bien de taille 64) en environ 10 minutes sur ma machine et après avoir exploré environ 1 milliard de noeuds. On peut aussi faire la comparaison sur fifty : on passe de 5 minutes avec astar à moins de 10 secondes avec idastar après avoir fait environ 13 millions d'appels à search. On visite ainsi plus de sommets avec idastar (13 millions plutôt que 4.5 millions), ce qui est normal puisque certains sont visités plusieurs fois, mais on consomme moins de mémoire.